デバイスの記事

SystemCによるビヘイビア合成入門

第3回 ビヘイビア合成の基本処理ステップ

桜井 至

今回は，ビヘイビア合成の処理手順について解説する．ビヘイビア合成では，ビヘイビア・モデルにクロック・サイクルを導入したり，クロック・サイクルをまたいで演算リソースを移動させることによって，面積や性能（クロック周期やレイテンシ）の最適化を図る．ビヘイビア合成の処理そのものは入力言語に依存しない．SystemCでもHDLでもほぼ同じ処理手順をとる．（編集部）

前回は，ビヘイビア合成で用いるSystemC構文について紹介しました．ビヘイビア合成で用いるSystemC構文は，BCA（bus cycle accurate；バス・サイクル精度）モデルに基づいて，ビヘイビア合成の対象となるC/C++アルゴリズムと外部システムを接続する入出力インターフェース，およびそれに伴うクロック・サイクル動作を定義するためのものと言えます．そのため，アルゴリズムと外部のハードウェア・システムを接続するラッパを作成することと見なすこともできます．

今回は，ビヘイビア合成の基本的な処理フローとそれぞれの処理ステップについて紹介します．

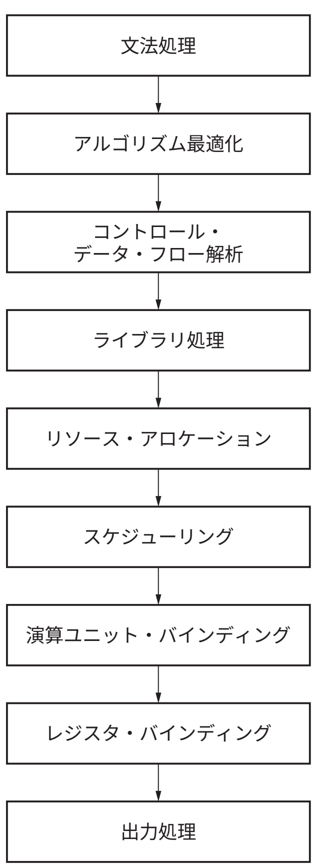


図1 ビヘイビア合成の基本処理ステップ
ビヘイビア合成の基本的な処理ステップを示す．それぞれのビヘイビア合成ツールによって処理順序や内容に若干の違いはあるが，基本的な処理ステップに大きな差はない．

● 一連の手順でビヘイビア記述をRTL記述に変換

図1に，標準的なビヘイビア合成の処理ステップを示します．それぞれのビヘイビア合成ツールによって処理順序や内容が異なる場合がありますが，基本的なステップにはあまり大きな差はありません．

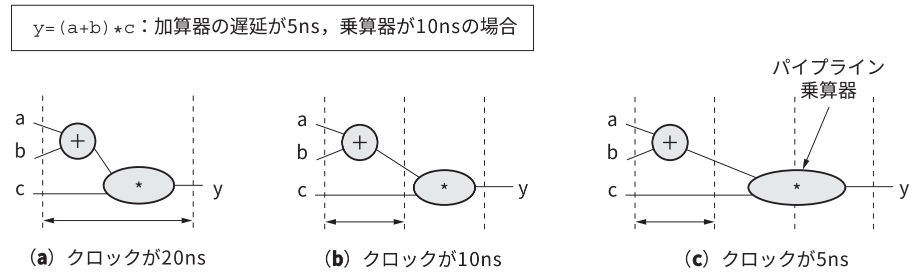
まずは，C++コンパイラに基づいた文法チェック処理と演算アルゴリズムの最適化を行い，入力記述のコントロール・データ・フローを解析します．論理合成でも同じような処理を行いますが，ビヘイビア記述はRTL（register transfer level）記述と比べて抽象度が高く，より少ないコードで機能を記述できます．

次に，演算子の実現に必要な演算リソース（加算器，乗算器など）の作成，またはライブラリからの選択を行い，リソース・アロケーション（resource allocation）を行った後，クロック周期やレイテンシなどのタイミング制約条件と，演算リソースの持つタイミング遅延値に従ってスケジューリングを実行します．

スケジューリングは，演算リソースの持つタイミングとクロック周期に依存します．例えば，**図2**に示したように

図2
タイミングによるスケジューリングの違い

スケジューリング結果は，クロック周期と演算リソースが持つ入出力遅延に依存する．y＝(a＋b)*cという式に対して，加算器の遅延が5ns，乗算器が10nsとすると，クロック周期20nsでは1サイクルで加算と乗算を実行できる．周期10nsでは加算と乗算がそれぞれ別のサイクルで実行され，2サイクルとなる．周期5nsではパイプライン乗算器が必要になり，合計3サイクルとなる．



リスト1　ビヘイビア合成を実行する記述例

```
typedef sc_uint<8> uint8;
sc_uint<33>  example_func(
    uint8 a, uint8 b, uint8 c, uint8 d, uint8 e )
{
  sc_uint<33> y;
  y = ( ( a * b ) + c ) * ( d * e );
  return y;
}
```

表1　SystemC/C++サポート構文

モジュール，スレッド
SC_MODULE
SC_CTHREAD
watching()
ポート
sc_in<>
sc_out<>
ポート・タイプ
bool
sc_int<>
sc_uint<>
sc_bigint<>
sc_biguint<>
sc_fixed<>

C++構文
すべてのC++演算子
配列メンバ/ローカル変数
if/else構文
for()ループ，while()ループ
switch()構文
ポインタおよび参照
クラスおよびクラス・テンプレートのインスタンス化
ファンクション・テンプレート
継承
演算子のオーバロード

```
y = (a + b) * c;
```

という式で，加算器のリソースの持つ最大遅延値が5ns，乗算器の最大遅延値が10nsであるとすると，クロック周期が20nsの場合，加算と乗算を一つのクロック・サイクルで実行できます．クロック周期が10nsであれば，加算と乗算を別々のサイクルで実行する必要があります．クロック周期が5nsであれば，乗算処理をパイプライン化したパイプライン乗算器を用いて合計3サイクルで演算を実行します．正確なスケジューリングを実行するには，正確な演算リソースの遅延値を求める必要があります．

次に，スケジューリングの結果に従って，演算リソースとレジスタに対する共有化とバインディング(binding)を行います．そして，最後にビヘイビア合成の出力データとして，データパス部とステート・マシン部で構成されたRTL記述を生成します．RTL記述をSystemCコードとして生成した場合，OSCI(Open SystemC Initiative；SystemCの標準化/普及推進組織)のリファレンス・シミュレータを用いてシミュレーションすることができます．また，RTL記述を論理合成可能なVerilog HDLコードまたはVHDLコードとして生成した場合，その後，論理合成を実行して，既存の設計フローへ進むことができます．

SystemCはクロックの概念を持っています．そのため，SystemCとHDLの混在が可能なシミュレータを使用した場合，同一のSystemCテストベンチを用いてビヘイビア検証とHDLによるRTL検証の両方を実行することができます．ビヘイビア・シミュレーションの実行結果とRTLシミュレーションの実行結果を比較して，機能が一致しているかどうかを確認できます[注]．

● ビヘイビア合成を構成する九つの処理ステップ

以下，各処理ステップに対してもう少し詳細なビヘイビア合成の手順を紹介します．また，実際の処理例として，**リスト1**に示した記述を含めて紹介します．

1) 文法処理

文法処理のステップでは，C++/SystemCのビヘイビア記述がビヘイビア合成可能かどうかをチェックし，データ構造をビヘイビア合成ツールが定義するフォーマットに合わせて変換して，内部データベースに格納します．文法処理では，一般的にC++コンパイラの技術を流用しています．

注：論理合成の前後でRTLの設計データとゲート・レベルの設計データを静的に比較する技術として，フォーマル・ベリフィケーション(形式的検証)が利用されている．ビヘイビア・レベルを対象とするフォーマル・ベリフィケーション技術は，まだ，研究段階にある．そのため，ビヘイビア合成の前後で機能が一致しているかどうかは，シミュレーションによって確認する必要がある．

OSCIのビヘイビア合成のサブ標準化委員会では，ビヘイビア合成可能な構文や記述スタイルの標準化を行っています．**表1**にビヘイビア合成可能な標準的なC++/SystemC構文を示します．ポインタ，参照，`class`，`struct`，`template`，オーバロードなどの構文がサポートされています（具体的なサポート構文と記述例は，次回以降で紹介していく）．

2）アルゴリズム最適化

アルゴリズム最適化のステップでは，記述内の定数の伝搬や保持，サブ式の共通化などの処理を行います．C++構文に対するいくつかの処理は，アルゴリズム最適化の段階で行われます．例えば，`for`ループや`while`ループなどのループ文は，そのままではインデックスごとに順序的に処理されてしまいます．しかし，ループの展開を指示することで，演算処理を並列化することができます．また，配列をメモリとして実現するか，ビットを展開してレジスタとして実現するかなどの処理が行われます．

3）コントロール・データ・フロー解析

コントロール・データ・フロー解析のステップでは，コントロール・データ・フロー・グラフ（CDFG：control data flow graph）を構築して，入力と出力の間の演算やデータの依存性の情報を抽出します．コントロール・データ・フロー・グラフは，出力がどのような手順に従って計算されるかを示す内部的なフロー・グラフで，タイミング情報は含みません．

リスト1に示した式からコントロール・データ・フロー・グラフを構築した例を**図3**に示します．出力がどのような手順や演算によって計算されるかを示すグラフが構築されています．この例では条件分岐が含まれていませんが，条件分岐がある場合は，条件分岐を含めたグラフが構築されます．

4）ライブラリ処理

ビヘイビア合成で生成したRTL記述を論理合成する際に，できるだけタイミング・エラーが生じないようにするため，ビヘイビア合成の段階からある程度タイミングを考慮する必要があります．なぜなら，ビヘイビア合成によって生成されたRTL記述が，論理合成時にタイミングの問題を起こしても，自動的に生成されたRTL記述を人手で修正することはほとんど不可能だからです．

データパス回路を構成する演算リソースについては，ターゲットのテクノロジを用いて論理ゲート・レベルのネットリストにマッピングすることで，ある程度正確なタイミングと面積を計算できます．論理ゲートで構成された演算リソースは，入出力遅延が小さければ面積が大きくなり，面積が小さければ遅延値が大きくなります．つまり，遅延値と面積の間にはトレードオフの関係があります．そのた

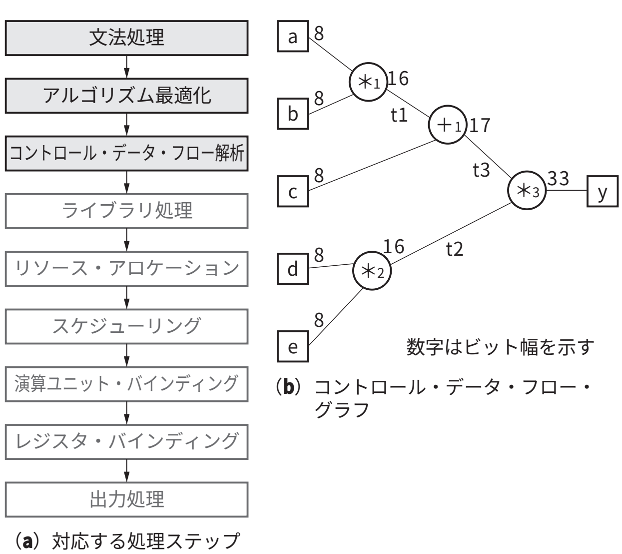
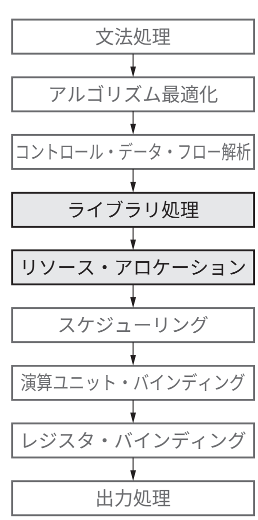


図3　コントロール・データ・フロー解析の結果例

演算式や条件文の構造を解析し，コントロール・データ・フロー・グラフ（CDFG）を生成する．リスト1に示した`example_func()`を解析した結果，図に示したようなCDFG構造が得られる．図中の数値は演算に必要なビット数を示す．条件分岐を含む場合，条件分けしたフローを含むCDFGが構築される．



（a）対応する処理ステップ

ファンクション	遅　延	面　積
乗算器： 8 * 8 =16	2.78ns	4,896
加算器：16+16=17	1.99ns	1,440
乗算器：20*20 =40	5.88ns	27,692

a*a = b，a+a = b：aは入力の，bは出力のビット幅を示す

（b）必要な演算リソースとその遅延，面積情報

演算リソース	個　数	面　積
8 * 8 =16	2	9,792
16+16=17	1	1,440
20*20 =40	1	27,692
合　計	4	38,924

（c）初期的なアロケーションによる面積の合計

図4　ライブラリ処理とリソース・アロケーション

図3の演算に対応して，論理ゲート・レベルで表現された演算リソースを作成するか，すでに用意されたライブラリから演算リソースを選択する．目標とするクロック周期が10nsとすると，それぞれのリソースの遅延値は10nsの条件を満足している．さらに，初期的なアロケーションによる合計面積を示す．

め，ライブラリとして，一つの演算子に対して高速タイプや小面積タイプなど，複数の演算リソースが用意されています．後処理のスケジューリングとバインディングでは，これらの演算リソースの中から所望の特性を持つものを選択します．

ライブラリ処理のステップでは，演算子に対応した演算リソースを作成したり，すでに用意された回路ライブラリの中から必要な演算リソースを選択したりします．**図4**に，**図3**に示したグラフを実現するために必要な演算リソースとその遅延値，および面積を示します．

5）リソース・アロケーション

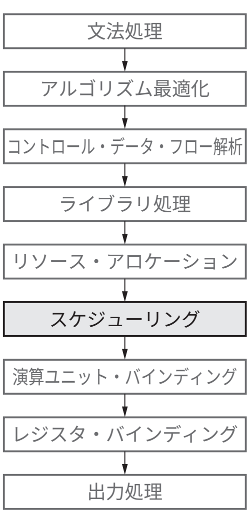
リソース・アロケーションのステップでは，演算子に対してそれを実現する演算リソースのアロケーション（割り付け）を行います．まず，スケジューリングの実施に必要な初期的なアロケーションを行います．この初期的なアロケーションで決まった演算リソースの割り付けは，以降の処理ステップであるスケジューリングやバインディングの段階で逐次変更されていきます．

6）スケジューリング

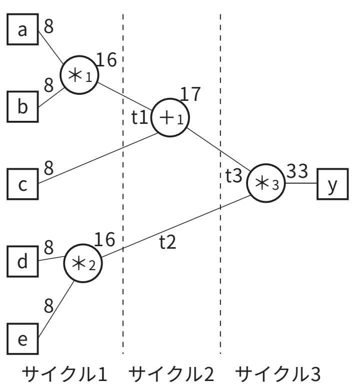
スケジューリングのステップでは，クロック周期やレイテンシ（クロック・サイクル数），スループットなどのタイミング制約をコントロール・データ・フロー・グラフに与えて，その制約を満たす範囲内で実現可能なクロック・サイクルを求めます．スケジューリングでは，設定された制約に対して最適な構成を見つけるために，記述コードの移動や再順序化を行います．スケジューリングの結果は，演算リソースを最小化しながら制約を満足する，サイクル精度のデータ・フロー・グラフになります．

ビヘイビア合成で与えるタイミング制約の情報は，ユーザが指定します．与えられるタイミング制約は，クロック周期だけでなく，必要に応じて，入力から出力までのレイテンシや入力信号の遅れ時間，要求される出力遅延などを指定します．

ビヘイビア合成と論理合成で大きく異なる処理が，スケジューリングです．スケジューリングによって，複数のRTL記述を容易に生成することができます．その結果，ビヘイビア合成には，RTL設計では非常に困難だった，クロック周期やテクノロジの変更を伴う設計資産の再利用が容易に行えるという利点があります．



（**a**）対応する処理ステップ



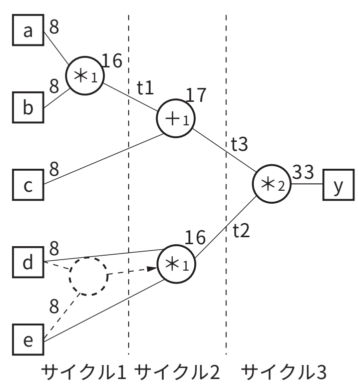
（**b**）コントロール・データ・フロー・グラフ

演算リソース	サイクル1	サイクル2	サイクル3	個　数	面　積
8＊8 =16	t1=a*b t2=d*e	—	—	2	9,792
16+16=17	—	t3=t1＋c	—	1	1,440
20*20=40	—	—	y=t2*t3	1	27,692
合　計				4	38,924

（**c**）処理結果

図5　最初のスケジューリングの結果例

3サイクルで実現するように制約を与えると，例えば図に示したような最初のスケジューリング結果が得られる．



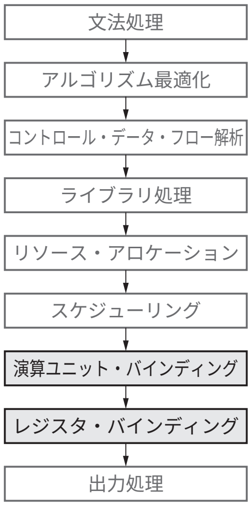
（**a**）コントロール・データ・フロー・グラフ

演算リソース	サイクル1	サイクル2	サイクル3	個　数	面　積
8＊8 =16	t1=a*b	t2=d*e	—	1	4,896
16+16=17	—	t3=t1＋c	—	1	1,440
20*20=40	—	—	y=t2*t3	1	27,692
合　計				3	34,028

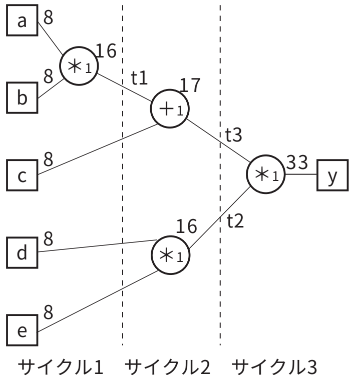
（**b**）処理結果

図6　演算リソースを移動したスケジューリングの結果

d*e演算をサイクル2に移動しても演算結果が変わらないので移動させた．これにより，サイクル1における同一サイクルの二つの8*8演算がなくなり，8*8の乗算器を一つ削除することができる．スケジューリングでは，演算子を実行するサイクルを変更することで，演算リソースの共有化を図る．



(a) 対応する処理ステップ



(b) コントロール・データ・フロー・グラフ

演算リソース	サイクル1	サイクル2	サイクル3	個　数	面　積
16+16=17	—	t3=t1+c	—	1	1,440
20*20=40	t1=a*b	t2=d*e	y=t2*t3	1	27,692
合　計				2	29,132

(c) 処理結果

図7　バインディングによる演算リソースの共有化

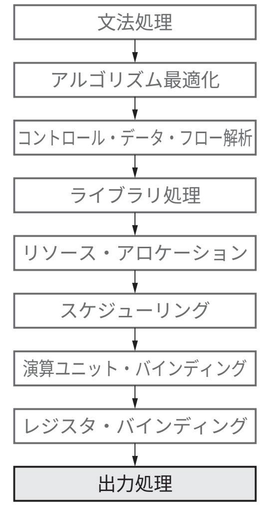
バインディング・プロセスでは，演算のファンクション・ユニットに対してリソース・コンポーネントを割り付ける．8*8の乗算に対して20*20の乗算器が使用可能なため，20*20演算をバインディングすることでさらに面積を削減できる．

図5に初期スケジューリングの結果を，**図6**に演算サイクルを移動させたスケジューリング結果を示します．**図6**では，サイクル1で実施していた二つの乗算のうちの一つを2番目のサイクルに移動することで，乗算リソースの数を減らしています．同じ機能を保ちながら，必要な演算子が最小になるように，演算の実行サイクルを移動させていくのです．

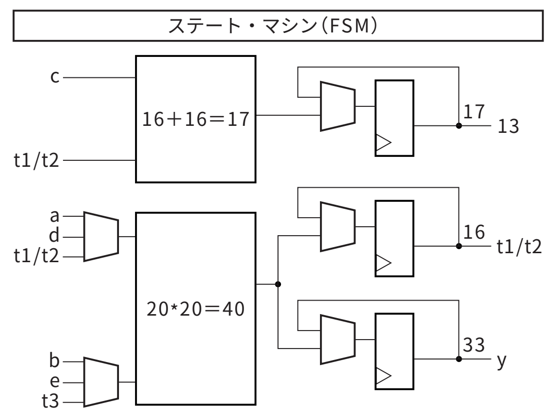
7）演算ユニット・バインディング

演算ユニット・バインディングのステップでは，スケジューリングされた結果に対して，ライブラリの中の演算リソースのどれを使用するのかを確定していきます．どのサイクルでどの演算を実行しているのかを判断し，異なるサイクル間の演算を共有化したり，可能な限り同一の演算リソースを用いるようにします．さらに，タイミングを満足する範囲で，より面積の小さい演算リソースに置き換えたりします．

図7では，8*8の乗算と20*20の乗算が同じサイクルで実行されていないため，8*8の乗算器の代わりに20*20の乗算器を使用することで乗算リソースの数を減らしています．



(a) 対応する処理ステップ



(b) 処理結果

図8　生成されたRTL記述の回路構造

データパス部と，データパス部のマルチプレクサを制御するステート・マシン部から構成されるRTL記述が生成される．

8）レジスタ・バインディング

異なるクロック・サイクルの間で使用する変数は，レジスタに保持されなければなりません．レジスタ・バインディングのステップでは，レジスタに値が書き込まれ，その値が実際に演算などに使用されるまでのクロック・サイクル（これを「ライフタイム」と呼ぶ）を解析し，その結果に基づいてできるだけレジスタを共有化していきます．レジスタのライフタイムをうまく組み合わせて，必要なレジスタ数を最小化します．

例えば，**図7**に含まれているt1，t2，t3の値は，レジスタへの書き込みが行われた次のサイクルで使用されるため，ライフタイムは1サイクルです．t2，t3は同じクロック・サイクルで書き込まれるため，共有化できません．しかし，t1とt2は異なるサイクルで使用されるため，共有化でき

COLUMN ビヘイビア合成で生成したRTL記述の品質は設計者が作成したRTL記述に匹敵するか

ビヘイビア合成で生成したRTL記述の面積や性能(クロック周期やレイテンシ)が，人手で作成したRTL記述のそれに匹敵するかどうかは，多くの設計者が興味を持っているポイントだと思います．しかし，この質問に対する回答は，Yes，Noで答えられるほど単純なものではありません．

筆者は，この質問に対して，「ハードウェアを意識してアルゴリズムを記述し，設計制約やRTLの構造を制御する属性やオプションを適切に指定すれば，人手で作成したRTL記述に匹敵するか，より性能の高いRTL記述を合成することができます」と答えています．どのようなビヘイビア記述を行っても，良いRTL記述が合成されるわけではありません．想定しているRTL設計の構造を意識して記述し，適切な設計制約条件やビヘイビア合成を制御する属性，およびコマンドを指定することで，初めて良いRTL記述が生成されるのです．

これは，HDLを用いた論理合成ツールの場合と同じだと思います．STARC(半導体理工学研究センター)が発行している「RTL設計スタイルガイド」では，「目標となる構造が明確な場合にはHDLでその構造を明確に定義する」と記載されています．これには，大きなalways(process)ブロックを構造が明確になるように複数のブロックに分割したり，中間変数を適切に使用したりすることが含まれます．(RTLの)HDLを適切に記述しないと，より良い性能を持つゲート・レベル回路を合成できないことは，今では常識です．

ビヘイビア合成の場合，HDL設計ほど詳細に構造を定義する必要はありません．しかし，適切なデータ幅の指定，冗長式の簡略化，並列処理の指定，配列やループの展開，関数の共有化などを行う必要があります．

ビヘイビア合成で生成したRTL記述と設計者が人手で作成したRTL記述の性能の比較は興味深い問題だと思いますが，これは，実際に比較できるRTLの設計資産が存在するから可能な議論であるとも言えます．論理合成ツールが実設計に使われ始めた当初，人手設計によるゲート・レベル回路との面積比較がよく行われました．しかし，比較するべきゲート・レベルの回路がまだない新しい設計プロジェクトに論理合成ツールが使われ始めると，面積比較はあまり話題にならなくなりました．

ビヘイビア合成に対する信頼が高まり，新しい設計プロジェクトに適用されるようになれば，既存のRTL記述との性能比較は問題にならなくなると筆者は考えています(もちろん，ツールに対して適切な制御がなされることが前提の話だが…)．現状のビヘイビア合成は，ようやく実用レベルに達してきた段階と言えます．まだまだ機能拡張が必要な部分が多くあります．

しかし，ビヘイビア合成を用いた大きなメリットである抽象度の向上による設計期間の短縮，検証の効率化，設計資産の再利用，ハードウェアとソフトウェアの協調設計などの効用が認知されるようになれば，徐々に，既存のHDL設計からの移行が進むと考えています．

ます．

9) 出力処理

出力処理のステップでは，アロケーションとバインディングの結果に基づいてRTL記述を生成します．SystemCのRTL記述を生成した場合は，SystemCのテストベンチといっしょに生成後のRTL記述をシミュレーションできます．一方，論理合成可能なVerilog HDLまたはVHDLのRTL記述を生成した場合は，論理合成およびレイアウトの工程へと進むことができます．SystemCとHDLの混在に対応したシミュレータがあれば，SystemCの環境の中で，生成したHDLのRTL記述をシミュレーションすることも可能です．

図7のバインディングの結果からRTL記述を生成したときのRTL設計の構造を**図8**に示します．図中のマルチプレクサのセレクト信号はステート・マシン部から出力され，ステート・マシンが制御する処理順序に従って演算が実行されます．

また，出力処理のステップではRTL記述の生成とともに，使用された演算リソースや面積，実現されたレイテンシなどのレポートが出力されます．

今回説明したビヘイビア合成の基本的なフローと処理ステップは，実はSystemCという言語そのものとは特に関係はありません(HDLのビヘイビア合成ツールも，処理内容はほとんど同じ)．ビヘイビア合成では，タイミング制約に従ってスケジューリングを実行し，クロック・サイクルを割り付けるということを理解していただければと思います．

さくらい・たかし
フォルテ・デザイン・システムズ(株)

<筆者プロフィール>
桜井 至．EDA業界に入り，HDLとしてはVerilog HDLから始まり，VHDL，Verilog-AMSへと進み，現在はSystemCを担当．さらに合成ツールとしては，論理合成，アナログ合成と進み，現在はビヘイビア合成を担当．設計言語と合成の大好き人間と言えるかもしれません．著書に「HDL設計入門」，「HDLによるディジタル設計の基礎」，「LSI設計の基礎技術」(テクノプレス社)，および「Verilog-AMS入門」(CQ出版社)がある．現在，フォルテ・デザイン・システムズ(http://www.forteds.com/)に所属．